SOFTWARE VERIFICATION SYSTEM, METHOD AND COMPUTER PROGRAM ELEMENT

Field of the Invention

This invention relates to security measures in computer software, and particularly but not exclusively to security verification of Java Virtual Machines.

Background of the Invention

In the field of this invention it is known that 'Java 2' includes a significantly enhanced security model, compared to previous Java Virtual Machines (JVMs). This new model can restrict the behaviour of a Java applet or application to a clearly defined set of safe actions. This allows a user to download application code from the internet or across a computer network, and run the code in a previously installed JVM. The user can be confident that the application will be assigned the required privileges to function correctly, but to neither damage the user's machine nor divulge sensitive information held on that machine to others via the network or internet.

However, a problem with this approach is that the JVM itself must retain its security integrity in order to ensure downloaded code is restricted in this way. If a malicious user (hacker) has been able to gain access to the user's machine outside of the JVM environment and alter the behaviour of the JVM the whole Java security model is undermined.

For example, the hacker could alter the privileges assigned for software code from a specific source, thereby allowing subsequently downloaded code from this source to function beyond the limits otherwise set by the JVM, and such enhanced privileges could easily be configured to compromise the security integrity of the user's machine.

Similarly, the hacker could disable the security code altogether, or worst still insert destructive routines into the core of the JVM which could be activated by an external trigger, such as specific time/date, or when other (possibly harmless) code is being executed.

It is clear that with this malicious activity, early detection of such a compromise of the JVM core would be very useful, and could prevent more serious subsequent damage.

If a malicious user decides to attack a machine, the JVM is an obvious target due to its significance in relation to web-based applications, servers and the like. Therefore the security integrity of the JVM is a highly significant factor in the security of the computer as a whole.

A need therefore exists for a software verification system, method and computer program element wherein the abovementioned disadvantages may be alleviated.

Statement of Invention

In accordance with a first aspect of the present invention there is provided a verification system for a computer software installation as claimed in claim 1.

In accordance with a second aspect of the present invention there is provided a verification method for a computer software installation as claimed in claim 7.

In accordance with a third aspect of the present invention there is provided a computer program element as claimed in claim 13.

Preferably the plurality of files include at least one tertiary file referenced by at least one secondary file of the plurality of secondary files, wherein after successful verification and selective loading of the at least one secondary files, the at least one secondary file is then used to manage the verification and selective loading of the at least one tertiary file.

The digital signature key is preferably a public key obtained via the internet. Alternatively, the digital signature key is preferably a hidden public key internal to the loader program, the loader program being arranged to use the hidden public key in the event that a public key cannot be obtained via the internet.

Preferably the digital signature key comprises a number of keys including a private key provided by an administrator, wherein the plurality of files includes at least one administrator-configurable file, wherein the loader program is further arranged to verify the digital signature of the at least one administrator-configurable file using the private key.

The software installation is preferably a Java Virtual Machine installation.

In this way a software verification system, method and computer program element are provided in which the security of the JVM is enhanced, a user has greater confidence that the Java applications will function correctly, and the detection of incorrect or damaged JVM installations is improved.

Brief Description of the Drawings

One software verification system, method and computer program element incorporating the present invention will now be described, by way of example only, with reference to the accompanying drawings, in which:

FIG. 1 shows an illustrative block diagram of a security methodology for files of a Java 2 software environment, in accordance with the present invention; and,

FIG. 2 shows an illustrative flow chart of a security procedure for the files of FIG. 1.

Description of Preferred Embodiment(s)

Referring to FIG. 1, there is shown an illustrative block diagram of computer software files associated with a Java Virtual Machine (JVM) installation, such as Java 2.

A third-party application 10 or a Java launcher 20 is used to initiate the JVM, and to load a primary library file, the JVM Dynamic link library (DLL) 30. This in turn loads other DLLs such as the secondary DLLs 40 and 60, and other configuration files such as the secondary 'config' file 50. In a similar way the secondary DLL 60 may in turn load other DLLs and configuration files, such as the tertiary DLL 70. Each of the above files in the JVM installation has a digital signature attached to it. Digital signatures are well known and well documented.

Binary files that will not be altered after installation, such as executables (.exe) and libraries (including the JVM DLL 30 and the secondary and tertiary DLLs 40, 60 and 70) will be digitally signed by the originating software house (or other trusted organisation). The signature will be stored in each binary file itself, in a manner well documented and well known in the art. The exact mechanism will depend upon the format of the file and how it is loaded by the operating system (OS). This is so that adding the signature does not invalidate the binary file and prevent it being loaded by the OS.

Configuration files such as the secondary config file 50 may be altered by the JVM administrator after installation (e.g., 'java.security', 'java.policy'), and these configuration files have a digital signature which can be updated by the administrator using the administrator's private key. The JVM is shipped with a utility which may be used by the JVM administrator to sign all configuration files using their own private key.

Referring now also to FIG. 2, the verification process starts at box 100. At box 110, a JVM loader is launched. This may be either the Java launcher 20 or indirectly via a

Java Native Interface (JNI) under the control of the third party application 10. Verification is similar in both situations; the verification of a JVM loaded by the launcher 10 will be described first.

The launcher application (Java launcher 20 or third-party application 10) seeks to obtain the JVM public key (to validate the signatures) from the internet site of the JVM provider, at box 120. If available, this is obtained (box 130). If the public key is not available a "hidden" copy of the key stored inside of the launcher is used (box 140) and a warning message printed to the screen (box 150). This allows for JVMs being used off line or behind fire walls, etc.

Whether obtained publicly (box 130) or from within the launcher (box 140), the public key is then used to verify the signature on the JVM DLL (box 160). If this is verified, the JVM DLL is loaded and control passed to it (box 180). If it is not verified, the JVM halts (box 170).

Once successfully verified, it is the responsibility of the JVM DLL to verify and load the secondary DLLs and configuration files used by the system, including the secondary DLL and config files 40, 50 and 60. At box 190, the JVM DLL looks for other files to load. If there are none, loading stops (box 200).

If there is another DLL or config file to load (such as the secondary DLL 60, the JVM DLL verifies the digital signature of that file (box 210) against the public or private key, by calling back to the JVM launcher 20. If the signature is not verified, that file is not loaded (box 220), and the JVM DLL continues with other required DLLs or config files (box 190), verifying each file in turn and only loading each file if verified successfully.

If the signature is verified, the file is loaded (box 230) and the JVM DLL continues with other required DLLs or config files (box 190), verifying each file in turn and only loading each file if verified successfully.

It will be understood that this "cascade" approach to verification can be used by subsequent DLLs loaded by the JVM DLL. Therefore once the JVM DLL has loaded all possible files, loaded DLLs such as the secondary DLL 60 may then seek to verify and launch other files. For example, the secondary DLL 60 may verify other files that it calls, following the procedure in boxes 190 to 230. In this way the secondary DLL 60 will verify the tertiary DLL 70, again calling back to the JVM launcher 20 to obtain the public key.

In the case of a virtual machine loaded via the JNI interface, for example by the third-party application 10 making a call to 'JNI_CreateJavaVM', the 'java/javac/..' launcher stage (box 110) will be bypassed. To allow the virtual machine to perform verification under these circumstances the third party developer can request a binary module from the JVM supplier to link into their application. This module will acquire the public key and verify the JVM DLL before loading using the same mechanism as described above.

It will be appreciated that the method described above for verification of the authenticity of a JVM using digital signatures will typically be carried out in software running on a processor (not shown), and that the software may be provided as a computer program element carried on any suitable data carrier (also not shown) such as a magnetic or optical computer disc.

It will be understood that the scheme for verification of the authenticity of a JVM using digital signatures described above provides the following advantages:

1. Firstly, the above arrangement provides enhanced security of the JVM.
2. Secondly, a user has greater confidence that Java applications will function correctly.

3. Furthermore, the detection of incorrect or damaged JVM installations is improved.

It will be appreciated that alternative embodiments to that described above are possible. For example, the order in which files other than the JVM DLL are loaded may differ from that described above. Furthermore, it will be appreciated that this invention is not limited to Java Virtual machine installations per se, but may find application in other similar software environments.